C/C++ Contributing Editors

The New C: X Macros

Randy Meyers

Assembly language programmers of the 60's had to develop some great tools just to preserve their sanity. Some of those tools, such as X macros, are still potentially useful today.

Sometimes a good idea just gets lost. Assembly language programmers of the 1960's invented a clever way to use macros to initialize parallel tables. This technique works perfectly well in C and C++, but seems almost unknown. The technique can also be used in some cases as a replacement for, or as an enhancement to, the C99 feature known as designated initializers.

Designated Initializers

Designated initializers permit you to name in the brace-enclosed initializer the particular array element or member being initialized when you initialize an aggregate or union. (See my April 2001 column for more details.) As such, designated initializers are a partial solution to the software engineering problem of maintaining an enum type definition and a parallel array giving the names of the enum literals:

```
#include <stdio.h>

enum COLOR {
  red,
  green,
  blue
};

char *color_name[] = {
  [red]="red",
  [green]="green",
  [blue]="blue"
};

int main() {
  enum COLOR c = red;
  printf("c=%s\n", color_name[c]);
  return 0;
}
```

Using designated initializers when initializing the array **color_name** has two advantages. First, it makes the initialization of the array independent of the order that the enum literals are declared in the definition of enum **COLOR**. Even if you rearrange the definition of the enum literals so that **red** has the integer value **2**, **green**

has the value **0**, and **blue** has the value **1**, the array **color_name** will have its elements initialized with the proper values in the right elements.

Second, since the names of the enumeration literals are explicitly used in the initialization of **color_name**, it serves as a clue that if you add a new enum literal to the definition of **COLOR** that you should add a new initializer to **color_name**. (A comment, of course, would also be helpful.) Unfortunately, too often adding a new enum literal to a large program requires searching the program source files for the names of the old enum literals to find all of the switch statements, arrays, and other places that need to be aware of the new enum name. Note that in a large program, it is likely that the definition of enum **COLOR** would be in a header file along with an **extern** declaration for array **color_name**, and the definition and initialization of **color_name** would be in a separate source file since it must be compiled only once. An explicit reference to an enum literal is thus much better than an implicit one when you have to search several source files for uses of the enum.

The use of designated initializers by **color_name** can also be a mixed blessing. Consider if a new enum literal **white** is added to **COLOR** between **red** and **green**, but no corresponding initialization is made to **color_name**. The array **color_name** will have the right number of elements since **[blue]**, the highest numbered element is still initialized. The elements **[red]**, **[green]**, and **[blue]** will be initialized correctly. However, the element **[white]** will have not been initialized and will default to a null pointer. (The use of designated initializers does not change the fact that you need not initialize all of the elements of an array, and compilers need not issue a message if you only initialize part of an array.) The program will work as long as you do not attempt to reference the name of **white**. Hence the mixed blessing: if you manage to put the program with this bug into production, you would probably prefer that it mostly work. However, during testing, you would probably prefer that the program fail spectacularly and obviously.

X Macros

A better solution would be to automatically obligate programmers to add the initializer when adding a new enum literal, and to provide a single place to add both, even if the enum definition was in a header file and the array initialization was in a separate source file. The assembly language programmers had a preprocessor-based solution to this problem.

Their solution depended upon features common to most preprocessors for most languages, and that are part of the traditional and standard preprocessors for C and C++. Those features are that macros can call other macros, that macros may be redefined, and that a nested macro call is not expanded until the macro that calls it is expanded during one of its calls. Consider a use of this technique:

```
#include <stdio.h>

#define COLOR_TABLE \
X(red, "red")       \
X(green, "green")   \
X(blue, "blue")

#define X(a, b) a,
enum COLOR {
  COLOR_TABLE
```

```
};
#undef X

#define X(a, b) b,
char *color_name[] = {
  COLOR_TABLE
};
#undef X

int main() {
  enum COLOR c = red;
  printf("c=%s\n", color_name[c]);
  return 0;
}
```

In this example, **COLOR_TABLE** is a macro that expands into a series of calls to a two-argument macro named **X**. At the time **COLOR_TABLE** is defined, the macro **X** has not been defined, but that is okay since the definition of macro **X** is not needed until **COLOR_TABLE** is actually used and expanded.

COLOR_TABLE is expanded twice by this program, and a different definition of the macro **X** is used by each expansion. The first time **COLOR_TABLE** is expanded, the macro **X** is defined to expand to its first argument followed by a comma, and thus the result of expanding **COLOR_TABLE** is a comma-separated list of the enum literals inside the definition of enum **COLOR**. After the expansion of **COLOR_TABLE**, I undefine the macro **X**, which prevents accidental use of the macro name and also allows **X** to be redefined later with a different body.

The second time **COLOR_TABLE** is expanded, the macro **X** is defined to expand to its second argument followed by a comma, which results in a comma-separated list of string literals that properly initialize the array **color_name**. Again, after the expansion of **COLOR_TABLE**, the macro **X** is undefined to prevent accidental uses of the name.

The macro **COLOR_TABLE**, and its use of "X macros," improves program maintainability in several ways. If you add a new line (X macro call) to **COLOR_TABLE** in order to declare a new enum literal, you are forced to add the initializer as well. You can reorder the lines in the table, or insert lines, or delete lines, and the **color_name** array will still be initialized properly. You have one location to edit to add a new enum literal even if the declaration of the enum type is in a header file and the initialization of the array is in a separate source file. For example, the definitions of **COLOR_TABLE** and enum **COLOR** could be in a header, and the source file that initializes the array **color_name** would include that header so it could expand **COLOR_TABLE**.

There is a potential problem with defining a macro that expands into a series of X macro calls. If the table gets big, you may exceed the compiler limit on the number of characters in a source line when defining the table macro or when expanding the macro. C90 only required that compilers accept lines with fewer than 510 characters after line continuation and macro expansion. C99 increases that limit to 4,095 or fewer characters. In practice, many compilers allow source lines much larger than the minimum limits, but it is wise to avoid defining very large macros.

To avoid the line length problem, you can put the X macro calls in a header file and include that header file in the places that would have expanded the table macro, as in the following.

```
// File: color_table.h
X(red, "red")
X(green, "green")
X(blue, "blue")

// File: main.c
#include <stdio.h>

#define X(a, b) a,
enum COLOR {
#include "color_table.h"
};
#undef X

#define X(a, b) b,
char *color_name[] = {
#include "color_table.h"
};
#undef X

int main() {
  enum COLOR c = red;
  printf("c=%s\n", color_name[c]);
  return 0;
}
```

Putting the X macro calls in a header file also has the advantage that you can use conditional compilation in building the table of X macro calls. For example, if some systems have the color **purple** instead of **red**, the header **color_table.h** could be:

```
// File: color_table.h
#ifdef NO_RED
X(purple, "purple")
#else
X(red, "red")
#endif
X(green, "green")
X(blue, "blue")
```

Note that there is no requirement that X macros take two arguments. The alert reader probably has realized that in the previous examples the second argument to the X macro was a quoted string whose value was the same as the first argument. The X macro calls in **COLOR_TABLE** could have taken only one argument (the enum name). When **COLOR_TABLE** was expanded to initialize **color_name**, the X macro could have been defined to use the stringize operator to create the needed string literal. On the other hand, if you have more that one parallel table indexed by an enum, you might use X macros that take more than two arguments. Since the X macro is defined immediately before expanding calls to it, and undefined immediately afterwards, you might have a table macro containing X macro calls with

two arguments and a different table macro containing X macro calls with five arguments. There is no conflict since a different definition of the X macro would be used when expanding the two table macros.

X macros might expand into something more complex than a copy of one of their arguments. Suppose we wanted to have a small gap in the value of our enum literals. We might use three-argument X macros as follows:

```
#define COLOR_TABLE \
X(green, , "green") \
X(red, =3, "red")   \
X(blue, , "blue")

#define X(a, b, c) a b,
enum COLOR {
  COLOR_TABLE
};
#undef X

#define X(a, b, c) [a]=c,
char *color_name[] = {
  COLOR_TABLE
};
#undef X
```

The second parameter in the X macros calls is an optional initializer for the enum literal itself to set its value. This example also shows the C99 feature wherein a macro argument is allowed to consist of no tokens. Such an argument expands into nothing in the macro body. Thus, the definition for enum **COLOR** is:

```
enum COLOR {
green, red=3, blue,
};
```

Since our enum literals now have gaps in their values, the X macro used during the initialization of **color_name** expands into designated initializers.

The above definition of enum **COLOR** shows another C99 feature I have been quietly using in this article. C traditionally allows a trailing comma in a bracketed initializer list as an aid to machine-generated source text. C90, however, did not allow a corresponding trailing comma in a enum literal definition list, although many C compilers allowed the comma as an extension. C99 now permits the trailing comma for enum literal definitions. As shown above, the X macro expansions in all of my previous examples generated a trailing comma after the last enum literal definition. If you are using a compiler that does not accept the trailing comma, you can comma-separate the X macro calls in the table macro and not produce a comma in the X macro expansion as in the following.

```
#define COLOR_TABLE \
X(red, "red"),       \
X(green, "green"),   \
X(blue, "blue")

#define X(a, b) a
enum COLOR {
```

```
  COLOR_TABLE
};
#undef X
```

Acknowledgments

The X macro technique was used extensively in the operating system and utilities for the DECsystem-10 as early as 1968, and probably dates back further to PDP-1 and TX-0 programmers at MIT. Alan Martin introduced X macros to me in 1984. I wish to thank Alan Martin, Phil Budne, Bob Clements, Tom Hastings, Alan Kotok, Dave Nixon, and Pete Samson for providing me with historical background for this article.

Randy Meyers *is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*